CHAPTER 4

Paradi m Shift

We hire people who build doghouses, give them cranes and ask them build skyscrapers. Then we're sur sed when they fail.

ileen Steets Quann

We said in Chapter 1 that if we are indeed at a juncture of discontinuous innovation, then we should be able to identify chronic problems with the current paradigm, as well as critical innovations that solve those problems at the cost of changes in accepted methods and practices. Having studied complexity and change, we are now ready for this analysis.

Chronic Problems

This chapter looks at the chronic problems that object orientation cannot solve and at critical innovations that can help solve them.

Like all paradigms, object orientation cannot solve certain problems. These are the chronic problems that demonstrate the limits of the paradigm. Some of them have resisted solution since the inception of the paradigm, while others have only recently come to light. In the Preface, we said that these problems fall into four categories: monolithic construction, excessive generality, one-off development, and process immaturity. We will look at problems in each of these categories in this section. As we shall see, many of these problems arise from tensions between methods used to deal with complexity and practices used to deal with change.

Monolithic Construction

We have not yet learned to assemble reusable components on a commercially significant scale.

It has long been recognized that one of the keys to productivity is building new components or systems by assembling existing ones. Development by assembly was one of the most important changes promulgated by the industrial revolution, enabling industries previously based on craftsmanship to meet the demands of an increasingly industrialized society. It was also part of the vision espoused by the early pioneers of object orientation, who predicted that new software would some day be assembled from existing objects. Unfortunately, we have not yet learned to assemble reusable components on a commercially significant scale. This is disappointing, since object orientation appears to be equipped to solve the problem. Its basic metaphor is encapsulation, which partitions a problem into opaque pieces, as we have seen. Of course, to be useful, partitioning must be accompanied by assembly. Unless we can assemble the pieces to produce a solution to the problem, we have gained nothing by encapsulation. Development by assembly should therefore be a natural consequence of the object-oriented paradigm.

Software is still produced almost entirely by construction from scratch.

We now know that objects are too fine-grained and context-dependent to be units of development by assembly. When Component Based Development (CBD) appeared, there was broad expectation that it would correct this problem by providing a coarser-grained and less context-dependent component model, fulfilling the vision with only a modest change to the basic paradigm. Unfortunately, despite the promise of CBD, software is still produced almost entirely by construction from scratch. Some observers blame the poor supply of reusable components, or the difficulty of finding the right components to solve a given problem. However, research and observation both suggest that these problems are symptoms of more fundamental problems, such as platform specific protocols, weak packaging, eager encapsulation, and the Not Invented Here syndrome. Let's look at each of these problems in turn.

MONOLITHIC

According to Wikipedia, a monolithic object is created in one piece, without any subcomponents. A monolithic program behaves as a single program, rather than as a collection of intercommunicating programs.

Platform Specific Protocols

It is difficult to assemble components across platform boundaries.

Current component assembly technologies depend too heavily on specific platforms, making it difficult to assemble components across platform boundaries, even when the boundaries in question are between different versions and/or configurations of the same products. CORBA and DCOM were the dominant technologies when CBD appeared. They were replaced, respectively, with Java Remote Method Invocation and .NET Remoting. While all of these technologies have succeeded in some ways, none of them has been able to support a component marketplace, or enable secure and efficient distributed interaction at the scales needed to automate business processes that span organizational boundaries. The problem is that they are closely tied to the technologies used to implement the components. This requires interacting components to be based on version compatible instances of the same implementation technologies. Components based on Java RMI, for example, do not interoperate well with components based on .NET Remoting or even with components based on different versions of RMI in some cases.

Weak Packaging

Weak specification and packaging technologies cause architectural mismatch.

Current component specification and packaging technologies do not capture enough information about the properties of components and about how they interact, making it hard to predict the properties of software developed by assembly. For example, given an arbitrary set of components, it is generally quite difficult to determine what they do, what dependencies they have, what resources they may consume, what performance they may exhibit, or what vulnerabilities they may create in an assembled system. When these unstated dependencies and assumptions conflict, architectural mismatches occur, making assembly difficult or impossible, or degrading the operational qualities of the resulting system. According to Garlan [GAO95], conflicting assumptions can be made about the components, the ways in which they interact, the structure of the system, and the processes used to produce an assembled system.

Mismatched assumptions about components can contribute to excessive code size and degrade the operational qualities of the software.

Assumptions about Components

Conflicting assumptions about components can include:

- *Which Supporting Services Are Required.* Components generally include or require the inclusion of

infrastructure that supplies services like security, persistence or transactions. Since comparable but different infrastructure may be supplied or required by other components, or by the underlying platform, redundant infrastructure packages may be included in the assembled system.

- *Which Components Are Responsible for Controlling Execution.* Components can make conflicting assumptions about which components will control execution. For example, some components use an event loop to organize interaction among modules, and provide access to services through call backs. If more than one of the components in an assembled system takes this approach, then adapters or mediators must be developed to allow the components to coexist effectively.
- *How Information Structures Should Be Manipulated.* Components can make conflicting assumptions about which components will be responsible for instantiating, copying or destroying other components. They can also make invalid assumptions about information structures maintained by other components, such as assuming that nested nodes in a tree can be added or deleted independently, while the tree control requires them to be added or deleted only by their parents.

These kinds of mismatches can contribute to excessive code size, or to problems with operational qualities, such as usability, reliability, performance, supportability, and especially security.

Assumptions about Relationships

Mismatched assumptions about the relationships between components can require extensive changes to the components or extensive adaptation or mediation.

Conflicting assumptions about the relationships between components can include:

- *How Related Components Will Interact.* For example, some components assume a broadcast pattern, in which components publish messages asynchronously to tell others about events or state changes, while other components assume a synchronous request/response pattern. These mismatches impose awkward requirements on components, such as requiring components that must be small and lightweight for other reasons to manage multiple asynchronous data streams.
- *How Information Will Be Communicated in Interactions.* For example, one component might assume synchronous interaction with discrete typed values as arguments, while another might assume asynchronous interaction with serialized information streamed between components.

These kinds of mismatches can require extensive changes tc the components or the construction of extensive adaptation o mediation code to bridge the differences between them.

Assumptions about Architecture

Mismatched architectural assumptions can cause reliability problems.

Conflicting assumptions can also be made about the architecture of the assembled system and about the presence or absence of components playing certain roles. For example, one component might assume that component interaction will be mediated by a transaction monitor, while another might assume that peers will interact directly. These kinds of mismatches can cause reliability problems, such as data corruptions and deadlocks in concurrent execution scenarios.

Assumptions about Process

Mismatched process assumptions can make builds labor-intensive and complex.

Conflicting assumptions can also be made about the process used to produce the assembled system. For example, a tool supplied by one component might generate distributed method invocation code from method signatures, while a tool supplied by another component might generate classes from models, requiring developers to write a script that extracts method signatures from classes generated by the second tool if they contain methods that will be invoked remotely. These kinds of mismatches can make builds complex and labor intensive.

Eager Encapsulation

Eager encapsulation makes component boundaries difficult to change.

Current component implementation technologies generally use eager encapsulation mechanisms, such as monolithic class definitions, that make component boundaries difficult to change. This means that components cannot be easily adapted during the assembly process to accommodate the differences between them, or to make them fit into different contexts. For example, dependencies on specific infrastructure, such as CORBA, are usually woven into the implementations of business components, making it impossible to use them with different infrastructure. Eager encapsulation intertwines the intrinsic or functional aspects of a component with the contextual or non-functional aspects, exposing them to changes in the contextual aspects.

There is a key tradeoff between encapsulation and agility.

There is a key tradeoff between encapsulation and agility. Encapsulation increases agility by reducing complexity. It is faster to extend a framework, for example, than to write equivalent functionality from scratch because many design decisions are made in advance by the framework. However, eager encapsulation limits agility because the intrinsic or functional aspects of a component cannot be easily separated from the contextual or non-functional aspects, making it hard to move them across component boundaries when refactoring is necessary to accommodate new requirements. Eager encapsulation is a minor concern when all of the source code can be modified, but it can be a major concern when the components come from different suppliers, or when some of them are only available at runtime, as in the case of Web services.

Not Invented Here Syndrome

It is more economical in the large to reuse imperfect components that already exist than to develop perfect new ones from scratch.

Finally, one of the most pervasive and insidious causes of monolithic construction is the infamous Not Invented Here (NIH) syndrome. Victims afflicted with this disease generally distrust anything developed outside their immediate organization, invariably choosing to build their own implementation of a required component rather than take a hard dependency on another organization by consuming an implementation that already exists. Sadly, victims also invariably underestimate the cost of developing, maintaining, and enhancing their proprietary implementation, usually by at least an order of magnitude. Of course, since no responsible decision maker would choose such an approach without a compelling reason, some fault must be found with the externally supplied implementation. While such grievances may be quite legitimate, more often than not, they are rather marginal. For example, a third-party XML parser might be seen as unusable because it is does more than is strictly required for the application in question, such as primitive type conversion, which has performance and memory footprint costs. Of course, experience in many other industries demonstrates that it is more economical in the large to reuse imperfect components that already exist than to develop perfect new ones from scratch.

The problem is with the immaturity of relationships between consumers and suppliers, and with the resulting lack of established practices and enabling technologies.

Before we vilify the victims of this disease much further, we should point out that their reluctance to take dependencies on external suppliers may not be completely unwarranted. Given the cost of finding existing components that can satisfy specific requirements, the cost of adapting them to new environments,

and the cost of finding defects in them once they have been ap plied, it may indeed be less costly and less risky to develop ne components from scratch than to reuse existing ones in specifi situations. In other words, the disease is often the result of hav ing been burned by suppliers who did not deliver on commi ments. Such experiences and the skepticism they produce ten to accompany ad hoc reuse, where consumers attempt to reus individual components opportunistically, in ways that wer never anticipated by their designs or implementations. Put an other way, the problem is not with the victims, but with th immaturity of relationships between consumers and supplier in the software industry. There is a lack of established prac tices and enabling technologies required to make reuse predic table, sustainable, and mutually beneficial for consumers an suppliers.

Changes are needed on an industrial scale to realign every day thinking and practice.

Of course, there is a bootstrapping problem. No project owne wants to push the envelope of established practice for the long term benefit of the industry, by trying to make reuse work o his project. The problem is too large for any project or any smal number of projects to solve because it requires the creation of market place for reusable components. Changes are needed o an industrial scale to realign every day thinking and practic about reuse. We will look at critical innovations that can brin about such pervasive changes later in this chapter, and we wil show how they can be applied successfully in Chapter 5.

Gratuitous Generality

We need general-purpose 3GLs only for a small part of the typical business application.

Current software development methods and technologies gen erally give developers more degrees of freedom than are nec essary for most applications. Most business applications, fo example, consist of a few basic patterns. They read informa tion from a database, wrap it in business rules, present it t a user, let the user act on it under the governance of thos rules, and then write the results back to the database. Of course this is an over simplification, since real world business appli cations generally contain a variety of additional challenges such as interactions with legacy systems, massive amounts o data, massive numbers of concurrent users, or severe quality of service constraints that make the implementation of thes basic patterns a lot more challenging than it sounds. We con tend, however, having developed many business applications that while there are always wrinkles that make each projec unique and specifics that always vary, such as the names of th

entities and the rule content, the vast majority of the work does not change much from project to project. Do we really need general purpose, Turing complete, third generation languages (3GLs) like C# and Java for more than a small part of the typical business application? Probably not.

There is much more benefit to be gained by moving to even higher levels of abstraction.

Recall from Chapter 2 that raising the level of abstraction reduces the scope of the abstractions and the amount of control over the details of the implementation given to developers. Recall, also, that the result of these losses is a significant increase in the power of the abstractions, enabling them to provide far more of the solution for a problem in the target domain than lower level abstractions. We pointed out there that business application developers would much rather use higher level abstractions, such as those provided by C# and Java, and by their accompanying frameworks, the .NET Framework and J2EE, than assembly language and operating system calls. Just as moving from assembly language to C# yields significant benefits for business application development, such as higher productivity, lower defect levels, and easier maintenance and enhancement, there is much more benefit to be gained by moving to even higher levels of abstraction.

Most of the work can be made faster and cheaper by using constrained configuration mechanisms.

We are not advocating the use of a closed-end language like a 4GL, nor the use of a single language for all aspects of development. We think business application developers will work with 3GLs for the foreseeable future. However, we think that the latitude afforded by 3GLs is needed only for a small part of the typical business application. Most of the work can be made faster, cheaper, and less risky by using constrained configuration mechanisms like property settings, wizards, feature models (discussed in Chapter 10), model-driven development, and pattern application.

If building business applications entirely with 3GLs is counter productive, then why is it the mainstream practice? We think the blame lies with three chronic problems of object orientation: weak modeling languages, weak metadata management, and weak code generation. Let's look at each of these problems more closely.

UML has succeeded as sketching medium, but not as a model-driven development or execution language.

Weak Modeling Languages

Using models to raise the level of abstraction and to automate development has been a compelling but unrealized vision

for 20 years. The single largest impediment to realizing th vision has been the promulgation of imprecise general-purpos modeling languages as both de facto and de jure standard Since it combined concepts from several object-oriented meth ods, the Unified Modeling Language™ (UML) became a ra lying point for a model-based approach to development. De spite a large number of UML tools, however, UML has no done much to raise the level of abstraction at which develope write source artifacts, or to automate development. Its primar contribution has been to design, since it lacks the focus an semantic rigor required to contribute to implementation, an most of that contribution has taken the form of informal model sketched on white boards. Most of the progress in automatin development has been based on programming languages an structured markup languages like XML and HTML. Fowle puts this eloquently, identifying three different goals espouse for UML: informal design, which he calls sketching; model driven development, which he calls blueprinting; and direc execution where models are interpreted. He claims that UM has succeeded in the first goal by establishing a widely use visual vocabulary, but not in the other two.

UML lacks the precision required to automate development.

One of the reasons for this failure is lack of precision. De velopment requires precise languages. While much has bee done to ensure the precision of general purpose programmin languages like C# and Java, much less has been done to ensur the precision of languages offering higher level abstraction UML and its derivatives have demonstrated over the course o many years that they are incapable of providing such levels o precision. This is not surprising, since they were designed fo documentation, not development. Unfortunately, those wh stood to profit have long held them to be the only legitimat languages for object-oriented modeling, and have supporte their claims with marketing dollars. Despite disappointing re sults, the industry has therefore been slow to see the problem with UML, and to invest in precise modeling languages de signed for development not documentation.

UML does not support CBD well.

Another reason is poor support for component-based devel opment concepts, such as component specification and assem bly. Rather than offer nothing, which might have been prefer able, UML provides many gratuitously different, incomplet and incompatible ways to model components, none of whicl accurately reflects the component architectures in common us today. By offering many ways to solve the same problem, i

creates confusion, especially since it does not apply the various mechanisms consistently.

UML is too general for real domains and has weak extensibility mechanisms that make it hard to focus.

A third reason, one that makes it wholly unsuitable for model-driven development, is its generality, combined with weak extensibility mechanisms. UML models anything and nothing at the same time. Tightly bound initially to programming language concepts in vogue at the time of its creation, UML was later given simple extensibility mechanisms similar to text macros, to give it some hope of modeling real domains. Since then, a vast number of features have been added, many of them driven through the crack in the door created by extensibility. Most of this work was undertaken without clear statements of requirements or adequate attention to conceptual partitioning. Due to the weakness of its extensibility mechanisms, efforts to focus UML on real domains have not been successful. The resulting semantic ambiguity and lack of organization have prompted a redesign, but the new version presents an even larger and more complex language that is just as ambiguous and disorganized as its precursor. As we show in Chapter 8 and Appendix B, UML falls short of providing the high fidelity domain-specific modeling capabilities required by the forms of automation described in this book. Its main contribution to automation has been the concept of a modeling language supported by a metamodel. Commercial model-driven development tools will be based on technologies designed for development, not documentation.

Weak Code Generation

CASE failed because code appeared only near the end of the project.

An earlier generation of products, called Computer Aided Software Engineering (CASE) tools, tried to use models to automate development. These products failed for the following reasons:

- Customers were uncomfortable spending most of their time on models, with the promise of code appearing only near the end of the project. This required enormous confidence in the tools, and in the longevity of their vendors.

Poor quality code was generated from platform-independent models.

- They promised to generate code for multiple platforms from platform independent models, allowing customers to preserve investments in the models as the platform technologies changed underneath them. Of course, a similar promise is offered by byte code languages, such as C# and Java, and by earlier generations of programming languages that provided some measure insulation from hardware and operating system changes. With a few notable exceptions, however, mos attempts at code generation from platform-independe models failed miserably. Unlike byte coded language compilers, CASE tools failed to exploit platform featur effectively, producing naïve, inefficient, least common denominator implementations.

Round-trip engineering did not work well.

- Their round-trip engineering features generally produced poor results. They tried to fill the gap betwe the models and the platform by generating large amounts of source code. The resulting complexity overwhelmed developers, causing them to abandon th models and write code by hand, at a certain point in th development process. Generic class modeling tools based on UML exhibit similar problems. After a few round trips, the synchronization falls apart, and the models are abandoned, making them ineffective as source artifacts.

Weak Metadata Integration

Metadata from tools can be used to automate the software life cycle.

The opportunity to automate more of the software life cycle b integrating metadata from various tools has long been reco nized. Information captured by requirements, for example, ca be used to drive test case definition and test harness develo ment. Test results can be used to drive defect record creatio and can be correlated with design information to scope d fects to specific components. Information about infrastructu dependencies can be used to validate component deployme scenarios and to drive automatic infrastructure provisioning i a target execution environment. Many other forms of autom tion are also possible, as described in Chapter 7.

CASE tools also failed to standardize life cycle metadata.

CASE tools also promised to standardize life cycle metadat but failed to deliver on the promise. Standardization attempt such as the AD/Cycle initiative led by IBM in the late 1980 were unsuccessful. These efforts made great progress in mode ing metadata, but because the models had to satisfy the requir ments of tools from multiple vendors, they were complex an could not be changed quickly as technologies and methodol gies changed because even minor changes required consens among the vendors.

Information managed by development tools is still poorly integrated across the life cycle due to tension between file and database resident information and due to attempts to standardize metadata implementation formats and repository architectures.

Information managed by development tools is still poorly integrated across the life cycle. At least two major sources of difficulty can be identified:

- One source of difficulty is the tension between file resident information like source code and database resident information like defect records. File resident information generally requires little if any infrastructure, and can be easily organized, viewed, and edited using simple tools, while database resident information generally requires much more infrastructure and can only be organized, viewed or edited using specialized tools. File resident information also fits more naturally into file-based development processes, such as build and source control. Database resident information, on the other hand, can be queried and updated much more easily in bulk. It also supports much richer relationships between pieces of information.
- Another source of difficulty is the failure to distinguish between metadata integration and metadata unification. Metadata integration requires only that the metadata suppliers and consumers to agree on common interchange formats. Metadata unification, on the other hand, requires them to share standard implementation formats, or even standard repository architectures. Metadata unification is not required, since metadata integration can support rich product interaction, as electronic commerce has amply demonstrated.

These observations lead to a discussion of the infamous repository issue that has burned many development tool vendors, some of them many times, but we will defer that discussion until Chapter 7.

One-Off Development

One-off development treats every software product as unique.

One-off development treats every software product as unique, despite the fact that most software products are more similar to others than they are different from them. Currently, most software product planning focuses on the delivery of a single version of a single product, despite the fact that returns on investment in software development would be far higher if multiple versions or multiple products were taken into account. Failing

to leverage commonality across products impedes the harvesting, classification, and packaging of knowledge, such as the documentation of prescriptive guidance, recommended practices and patterns, or the development of reusable components.

Most development processes are designed to support one-off development.

The assumption of one-off development is so deeply ingrained in this industry that most of us are not conscious of either the practice or the alternatives. Most widely used development processes, whether formal or informal, are designed to support the development of a single product in isolation from others. This affects both their overall structure and their content. For example, both XP and UP assume that every product will be designed and developed independently from first principles to accommodate its unique requirements, and neither of them provides guidance for supplying or consuming reusable assets, or facilitates communication between consumers and suppliers of reusable assets.

One-off development permits only ad hoc reuse and fosters organizational dementia.

One-off development permits only an ad hoc and opportunistic approach to reuse, since there is no context for predicting what might be reused, or the situations in which the reuse might occur. It also creates a phenomenon called organizational dementia. An organization afflicted by this disease cannot capture knowledge in reusable forms. It therefore cannot leverage experience or lessons learned in subsequent efforts, and must constantly rediscover what it already knows. Documenting patterns and practices is a step in the right direction because it focuses on capturing and reusing knowledge across multiple versions or multiple products. Later in this chapter, we will consider a critical innovation that goes much further in the same direction.

Process Immaturity

Few organizations can develop satisfactory software on schedule and within budget.

One of the most challenging problems in software development today is managing the development process, especially when multiple groups must collaborate to develop a single product. According to Booch, a development process is a collection of practices designed to promote high quality, efficient resource allocation, and effective cost and time management. While methods vary widely, as we have seen, most organizations use some form of process, and yet few of them can consistently develop software that satisfies requirements on schedule and within budget. This problem suggests that current methods and practices are immature.

Most development processes tend to one of two extremes.

Most development processes in use today tend to one of two extremes: either they use excessive formalism to promote cross-group collaboration at the cost of agility, or they throw formalism to the wind and permit excessive autonomy to promote agility at the cost of cross-group collaboration. As we will show here, these extremes reflect a tension between the need to deal with complexity and the need to deal with change. Later in this chapter, we will suggest an approach that helps organizations deal with complexity, while preserving their ability to respond rapidly to change.

Excessive Formalism

Formal processes are optimized for dealing with complexity.

We said earlier that formal processes are optimized for dealing with complexity. By defining the development process in detail, formalism provides structure that helps the participants organize, understand, and reason about the software, development tasks, the development environment, and development organization. This kind of structure is absolutely essential when multiple groups must collaborate to develop a single product. In such an environment, changing an interface on which other groups depend can break the build, and changing the behavior of a component on which they depend can create usability issues, defects, security holes or performance problems. Changes to shared interfaces and components must therefore be carefully orchestrated. This orchestration prevents all groups from freely changing the software, and therefore limits their agility. It also requires the kind of structure described earlier. Collaborating groups must have a common understanding of shared components, common expectations as to how and when they might change, and common policies regarding tools, directory structure, file formats, and many other parts of the development environment.

Formal processes are often overly prescriptive.

Unfortunately, the industry has a tendency to become overly prescriptive, assuming that developers are naïve and uninformed, and attempting to use formal processes to prevent them from making mistakes by telling them what to build, how to build, and what skills are required to perform certain tasks. Examples of excessive formalism abound, and often involve heavy-handed application of the UP. Another problem with formality is sheer size and weight. Even though most of the material in a process like the UP is relevant to some project at some time, its scope is daunting. Although the material is well

organized, using it can require a serious time commitment b cause of the quantity. It can take much longer than expected sift through it, find bits relevant to a problem and apply the There are many cross-references that must be checked to det mine whether or not they are required for a given situation. we shall see, this kind of experience results from naïve app cation of the UP, and can be avoided through customization

Excessive Autonomy

Informal processes are optimized for dealing with change.

Frustration with the glacial pace of development that accor panies excessive formality spawned a global revolt among d velopers near the turn of the millennium, creating a moveme complete with a manifesto, recognized leadership, and annu conferences. Of course, we are describing the agile develo ment movement. Agile methods are the epitome of informa ity. We said earlier that informal processes are optimized f dealing with change by:

- Keeping the team small, which allows it to rely on verb communication, reducing the need for specifications.
- Working in small iterations, which keeps the software building and running, helping the team interact with business stakeholders to validate requirements.
- Refactoring constantly reduces fatigue, which makes th software more pliable and subsequent changes easier.

Agile methods work well for small projects, but not for larger ones.

Agile methods work exceptionally well for small projec staffed by small teams of expert developers that can work a tonomously. They create significant problems, however, wh applied on larger scales. Trying to go faster by increasing agili is like trying to make a company more profitable by cuttir costs. Both work well up to a point, and improve the health the organization, but then they have the opposite effect, as ne essary infrastructure is jettisoned along with the unnecessar Berrisford [Ber04] observes that highly agile parallel develo ment increases architectural mismatch, because teams work independently as possible. Making the teams agree on a con mon architecture helps ensure that the components they d velop can be integrated without extensive changes, adaptatic or mediation, but it reduces agility at the same time by invol ing more people, requiring more documentation, and forcir teams to synchronize their iterations. It also creates inflexib

requirements, since none of the teams sharing a component can change its interfaces without consulting the others. Likewise, refactoring is also harder, since shared interfaces become boundaries that cannot be crossed without consultation.

Adhering to agile development principles when scaling up can create problems.

Of course, refusing to relax some of the principles of agile development when faced with a requirement to scale up can create the opposite problems. Not agreeing on a common architecture ensures that there will be problems with architectural mismatch and unpredictable results when components from multiple teams are assembled, and that extensive changes, adaptation and/or mediation, will be required. It also ensures that the system will exhibit operational quality problems, such as poor usability, high defect levels, security vulnerabilities, and poor performance.

Agile methods can create supportability problems because agile teams do not produce much design documentation.

Agile methods can also create supportability problems because agile teams usually do not produce much design documentation. The lack of documentation makes it hard to determine after the fact what design decisions were made, what alternatives were considered, and what rationale was used to make the decision. Of course, if maintenance and enhancement are ongoing, and the team does not lose any members, then the knowledge required to keep up the software may still be fresh in the minds of the team members. There is significant risk in relying on memory alone, however, since people forget details with the passage of time, and change projects fairly frequently. While the code provides some documentation, there is also significant risk in relying on code alone, since code quality varies widely, and since source code contains many low-level details and accidental complexities that blur its essential behaviors and properties. Thus, a new team called upon to maintain or enhance the software may not be able to understand it quickly enough to maintain it effectively. These conditions can lead to yet another form of NIH syndrome, where the new team decides that it is better to rewrite a component than to try to maintain it. While rewriting may not be entirely unwarranted in some cases, it is usually a somewhat specious conclusion. Moreover, the cost of rewriting a non-trivial component is usually underestimated by at least an order of magnitude.

Geographical distribution can exacerbate these problems.

Geographical distribution can exacerbate these problems, not because people forget or move on, but because they are not available when needed, or because technology limitations and time zone differences restrict high bandwidth communication. We have come to appreciate the value of documentation after

leading projects involving teams in Europe and the US We Coast. Given the timezone differences, opportunities for inte active communication were limited and documentation helpe make the most of the time we did have together.

Critical Innovations

We consider critical innovations in four areas: systematic reuse, development by assembly, model-driven development, and process frameworks.

In this section, we describe critical innovations that solve thes chronic problems by taking new ground in the struggle again complexity and change. As we said in the Preface, we hav noticed that some of the most significant gains have been pr duced by a recurring pattern that integrates four practices t automate software development tasks. A framework is deve oped to bootstrap implementations of products based on a con mon architectural style. Products are then created by adaptin configuring, and assembling framework components. A lan guage to support this process is defined and supported b tools. The tools then help developers engage customers an respond to changes in requirements rapidly by building sof ware incrementally, keeping it running as changes are mad and capturing design decisions in forms that directly produc executables. We therefore consider critical innovations in fou areas suggested by the pattern: systematic reuse, developmer by assembly, model-driven development, and process fram works. According to the characterization of paradigm shif provided in Chapter 1, these innovations should build on th strengths of object orientation while addressing some of th weaknesses that gave rise to the chronic problems describe earlier.

Systematic Reuse

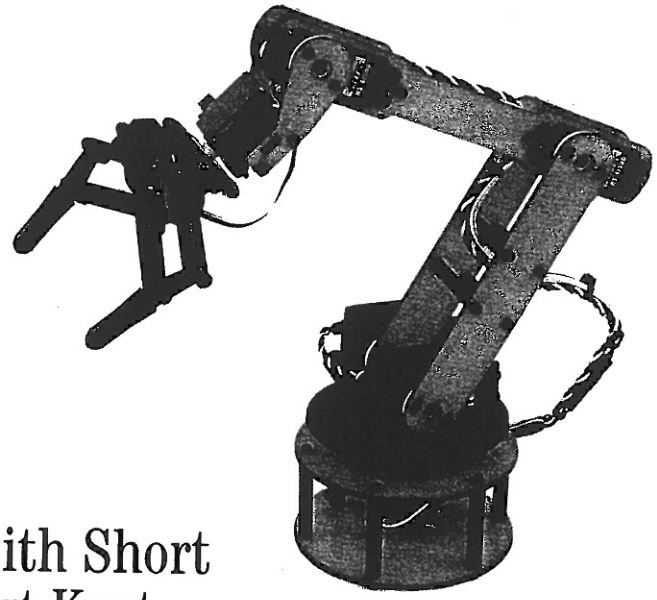
It is hard to make a component reusable in arbitrary contexts.

We saw earlier that an ad hoc and opportunistic approach t reuse usually falls short of expectations. The main problen with such an approach is lack of context for reuse. It is cha lenging, if not impossible, to determine what functionality component should possess and how it should be implemente to be reusable in arbitrary contexts. It is much easier to buil components that are reusable in the context of a specific do main or family of systems. For example, the .NET Framewor defines a family of rich client user interfaces on the Microso Windows platform. It is much easier to build reusable use

WILEY

Software Factories

Assembling Applications with Patterns, Models, Frameworks, and Tools



Jack Greenfield and Keith Short
with Steve Cook and Stuart Kent
Foreword by John Crupi